Measuring Various Properties of Execution Traces to Help Build Better Trace Analysis Tools

Abdelwahab Hamou-Lhadj
University of Ottawa
SITE, 800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
ahamou@site.uottawa.ca

Timothy C. Lethbridge
University of Ottawa
SITE, 800 King Edward Avenue
Ottawa, Ontario, K1N 6N5 Canada
tcl@site.uottawa.ca

Abstract

Understanding the behavior of a software system by studying its execution traces can be extremely difficult due to the sheer size and complexity of typical traces. In this paper, we propose that if various aspects that contribute to a trace's complexity could be measured and if this information could be used by tools, then trace analysis could be facilitated. For this purpose, we present a set of simple and practical metrics that aim at measuring various properties of execution traces. We also show the results of applying these metrics to traces of three software systems and suggest how the results could be used to improve existing trace analysis tools.

1. Introduction

Understanding the behavioral aspects of a software system can be made easier if dynamic analysis techniques are used.

Research in this area has led to the creation of many tools for rapid exploration and analysis of large execution traces [2, 8, 9, 13, 14, 15]. Using these tools, the analyst can perform various tasks including searching for specific events that occur in the trace, grouping selected events in the form of clusters, and detecting patterns of execution. These tools differ mainly with respect to the visualization techniques used as well as the way similarity is measured to implement the pattern detection capabilities.

However, after analyzing several execution traces, we observed that the behavior embedded in the traces can sometimes be considerably more complex than expected and that the techniques supported in most existing tools did not always help. This is partially due to the fact that none of these tools provides efficient guidance with respect to how to combine the supported features in order to effectively reduce the size of traces. For example, we would have liked to be able to know, in advance of studying a trace, or part of a trace, how much work might be involved in understanding it. This typically includes measuring several aspects of the trace such as the number of distinct components of the system that are invoked in the trace, the structure of the trace events, etc, something that is not supported by most existing tools.

We attribute this limitation to a lack of a set of practical metrics for measuring different aspects of an execution trace. In this paper, we present a set of metrics that can be implemented in trace analysis tools to facilitate the analysis of large traces. These metrics are designed based on the GQM [1] paradigm. In addition to this, we report the result of measuring various properties of several execution traces of three software systems. The outcome of this study can be used in different ways:

- The designers of trace analysis tools can design facilities to reduce the amount of information being displayed to some threshold (by hiding sufficient detail). For example, if the user selects the 'detection of patterns' feature then the tool might suggest to display patterns that occur more frequently than a certain threshold. The display of all possible patterns might be overwhelming and sometimes meaningless. Tool designers could also 'color' each subtree of a trace to give software engineers a better sense of the complexity to be found in that subtree before the software engineer 'opens' it for exploration.
- Researchers in the field of dynamic analysis (with a focus on program comprehension) can use these results to investigate further techniques for reducing the complexity of traces that will not rely heavily on the intervention of users.
- Software engineers exploring execution traces will be given the possibility to measure properties of traces in order to choose which traces to analyze, to generate traces that are neither too complex nor too simple, and to select parts of traces to analyze that have a suitable complexity level.

The execution traces targeted in this paper are those based on routine invocations. We use the term 'routine' to represent functions, procedures or methods of classes. Although the systems analyzed in this paper are object-

oriented, the metrics proposed here can also be applied to non OO systems.

This paper is organized as follows: In the next section, we introduce the concept of comprehension units that will help us determine some of the metrics used in this paper. In Section 3, we describe the metrics and motivate why they are important for characterizing the effort required to understand an execution trace. In Section 4, we show the results of analyzing traces of three software systems. We will use the outcome of this analysis to discuss the obstacles of current trace analysis techniques. In Section 5, we discuss how these metrics can be supported by tools.

2. The Concept of Comprehension Units

A trace of routine invocations can be represented using a tree structure as illustrated in Figure 1a. Traces usually contain several repetitions, which are either contiguous or non-contiguous. This is reflected in the tree as a repetition of the same subtrees.

We define a *comprehension unit* as a distinct subtree of the trace. We hypothesize that in order to *fully* understand the trace, without any prior knowledge of the system, the analyst would need to investigate all the comprehension units that constitute it. It is important to notice that in practice, full comprehension would rarely be needed because the analyst will achieve his or her comprehension goals prior to achieving a full comprehension. Also, the analyst will likely not need to try to understand the differences among the many comprehension units that only have slight differences.

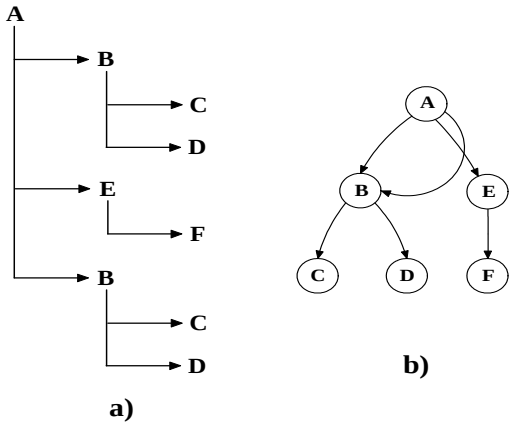

Figure 1. The graph representation of a trace is a better way to spot the number of its distinct subtrees.

An efficient technique for extracting comprehension units is based on transforming the trace into its compact form by representing similar subtrees only once. This transformation results in a directed ordered acyclic graph. This technique has been widely used for trace compression and encoding [10, 12] and was first introduced by Downey et al. [3] to enable efficient analysis of tree structures.

Figure 1 shows a trace T in the form of a tree structure and its corresponding directed ordered acyclic graph. The graph shows clearly that T contains 6 comprehension units and that the comprehension unit rooted at B is repeated twice in a non-contiguous way. We refer to comprehension units that are repeated non-contiguously as *trace patterns*.

3. Metrics

In this subsection, we present the metrics that are used in this paper to measure relevant properties of execution traces. We motivate the choice of these metrics using the Goal/Question/Metric model [1] as a framework. The goal can be stated as follows:

Enable software engineers to more quickly understand the behavior of a running system.

Software engineers will benefit from these metrics only if they are incorporated into tools, in ways such as those suggested earlier.

We have designed questions that aim at characterizing the content of an execution trace. We present these questions along with the metrics that address them.

3.1. Call Volume Metrics

This category of metrics aim at answering the following question:

Q1. How many calls (or invocations) does a trace contain?

Knowing the number of calls in a trace is one factor that will help determining the work required to understand the trace. The following metrics make this more precise:

Initial size [S]: The initial size is the raw number of calls made during the execution of the system. This forms a baseline for subsequent computations.

Size after removing contiguous repetitions [Scr]: This is the number of lines in the trace after removing contiguous repetitions due to loops. In other words, many identical calls are mapped into a single line by processing the trace. We refer to this process as the *repetition removal stage*.

According to our experience working with many execution traces, the number of lines after repetition removal is a much better indicator of the amount of work that will be required to fully understand the trace. This is due to the fact that the initial size of a trace, S, is highly sensitive to the volume of input data used during the execution of the system. To understand the behavior of an algorithm by analyzing its traces, it is often just as

effective to study the trace after most of the repetitions are collapsed. And, in that case, studying a trace of execution over a very large data set would end up being similar to studying a trace of execution over a moderately sized data set.

Collapse ratio after removing contiguous repetitions [Rcr]: This is the ratio of the number of nodes after removing the contiguous repetitions to the initial size of the trace. That is: Rcr = Scr/S.

Knowing that a program does a very high proportion of repetitive work (that Rcr is low) might lead program understanders to study possible optimizations, or to reduce input size. Knowing that Rcr is high would suggest that understanding the trace fully will be time consuming.

The Collapse Ratio is similar to the notion of 'compression ratio' that one might talk about in the context of data compression. However we have carefully avoided using the term 'compression' since it causes confusion: The purpose of compression algorithms is to make data as small as possible; a *decompression* process is required to reconstitute the data. On the other hand, the purpose of collapsing is to make the data somewhat smaller by eliminating unneeded data, with the intent being that the result will be intelligible and useful without the need for 'uncollapsing'.

3.2. Component Volume Metrics

This category of metrics is concerned with measuring the number of distinct components that are invoked during the execution of a particular scenario. The motivation behind using these metrics is that traces that cross-cut many different components of the system are likely to be harder to understand than traces involving fewer components. More specifically, these metrics aim at investigating the following question:

Q2. How many system components are invoked in a given trace?

The term 'component' is very general. Separate metrics can be used to measure invocations of different types of components. In this paper, since the target systems that are analyzed are all programmed in Java then it may be useful to measure the following:

Number of packages [Np]: This is the number of distinct packages invoked in the trace. By 'invoking' a package we mean that some method in the package is called.

Number of classes [Nc]: This is the number of distinct classes invoked in the trace

Number of methods [Nm]: This is the number of distinct methods that are invoked in the trace

It may be useful to create similar metrics, e.g. based on the number of threads involved. It may also be useful to know the proportion of a system that is covered by the trace. The more of a system covered, the more time that may be required to understand the trace, but the more complete an understanding of the entire system may be gained. Thus we measure the following:

Ratio of number of trace packages to the number of system packages [Rpsp]: This is the ratio of the number of packages invoked in a trace to the number of packages of the instrumented system. More formally, let us consider:

- NSp = The number of packages of the instrumented system
- Np = The number of distinct packages invoked in the trace

Then: Rpsp = Np/NSp

Ratio of number of trace classes to the number of system classes [Rcsc]: This is the ratio of the number of classes invoked in a trace to the number of classes of the instrumented system. This is computed analogously to Rpsp.

Ratio of number of trace methods to the number of system methods [Rmsm]: This is the ratio of the number of methods invoked in a trace to the number of methods of the instrumented system.

3.3. Comprehension Unit Volume Metrics

We showed earlier that a better representation of a trace consists of using its compact form (i.e. ordered directed acyclic graph). This category of metrics explores the ordered directed acyclic graph to compute several metrics. The question that we are interested in investigating is:

Q3. How many comprehension units exist in a trace?

We suggest that metrics based on comprehension units will give a more realistic indication than call volume of the effort required to understand a trace.

Number of comprehension units [Scu_{sim}]: This is simply the number of comprehension units (i.e. distinct subtrees) of a trace. However, this number may vary depending on the way similarity between the tree subtrees is computed. The subscript 'sim' is used to refer to the similarity function that is used.

Considering exact matches only may result in a very high number of comprehension units. For example, consider the tree of Figure 2; the two subtrees rooted at B differ only because the number of contiguous repetitions of their subtrees differs as well.

If the exact number of repetitions is ignored then we can compute Scu_{cr} to refer to the number of comprehension units that result after ignoring the number of contiguous repetitions. Similarly, other metrics can be computed using different matching criteria.

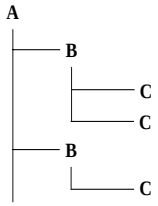

Figure 2. The two subtrees rooted at B can be considered similar if the number of repetitions of C is ignored.

Tree to graph ratio [Rtg_{sim}]: This metric measures the ratio of the number of nodes of the ordered directed acyclic graph to the number of nodes of the trace. We expect to find very low ratio since the acyclic graph represents repetitions only once. In previous work, we showed that this transformation results in a very high degree of collapsing of the trace [5]. More formally, let us consider:

- Scr = The size of a trace T after removing contiguous repetitions
- Scu_{cr} = The size of the resulting graph after transforming T with contiguous repetitions ignored.

Then: $Rtg_{cr} = Scu_{cr}/Scr$

If Rtg_{sim} is low then this suggests that even a huge original trace might be relatively easy to understand.

3.4. Pattern Related Metrics

This category of metrics is concerned with measuring the number of patterns that exist in a trace. The motivation behind computing these metrics is that patterns may play an important role in uncovering domain concepts or interesting pieces of computation that the understander would benefit from understanding [8, 9]. In this category, we aim at investigating the following question:

Q4. How many patterns exist in a trace?

We present the following metrics:

Number of trace patterns [Nptt]: This metric simply computes the number of trace patterns that are contained in a trace. As defined previously, a pattern is a comprehension unit that is repeated in a non-contiguous way. If Nptt is small this suggests that the complexity of the trace will be high.

Ratio of the number of patterns to the number of comprehension units [Rpcu]: This metric computes the ratio of the number of patterns to the number of comprehension units. In other words, we want to asses the percentage of comprehension units that are also patterns.

4. Analysis of Sample Traces

We analysed the execution traces of three Java software systems: Checkstyle [4], Toad [7] and Weka [16]. This analysis has the following objectives:

- Compute the metrics presented in the previous section
- Perform further analysis to interpret the measurements using the metrics
- Draw inferences about the applicability of the metrics by comparing the results from the three systems

4.1. Target Systems

To begin with, we describe the three target systems used in this study. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard [4]. This is very useful to projects that want to enforce a coding standard. The tool allows programmers to create XML-based files to represent almost any coding standard. Toad is an IBM tool that includes a large variety of static analysis tools for monitoring, analyzing and understanding Java programs [7]. Although these tools can be run as standalone tools, they can provide a much greater understanding of a Java application if they are used together. Weka is a collection of machine learning algorithms for data mining tasks [16]. Weka contains tools for data pre-processing, classification, regression, clustering and generating association rules.

Table 1. Characteristics of the target systems

	Packages	Classes	Methods
Checkstyle	43	671	5827
Toad	68	885	5082
Weka	10	147	1642

Table 1 summarizes the characteristics of the target systems. There are different ways of computing the size of an object-oriented system. In our case, we are only interested in the components that we will use to compute the metrics related to the trace content. We deliberately ignored the number of private methods (including private constructors) due to the fact that they are only used to implement a portion a large behaviour, and are restricted to the class that defines them. Our instrumentation does

not capture them either as we will show in the next subsection. Abstract methods are also excluded since they have no presence at run time. Finally, the number of classes does not include interfaces for the same reason.

4.2. Generating Traces

We used our own instrumentation tool based on BIT [11] to insert probes at the entry and exit points of each system's non-private methods. Constructors are treated in the same way as regular methods. Traces are generated as the system runs, and are saved in text files. Although all the target systems come with a GUI version, we can invoke their features using the command line. We favoured the command line approach over the GUI to avoid encumbering the traces with GUI components. A trace file contains the following information:

- Thread name
- Full class name (e.g. weka.core.Instance)
- Method name and
- A nesting level that maintains the order of calls

We noticed that all the tools use only one thread, so we ignored the thread information.

Table 2. Checkstyle Traces

Trace	Description
C-T1	Checks that each java file has a package.html
C-T2	Checks that there are no import statements that use the .* notation.
C-T3	Restricts the number of executable statements to a specified limit.
C-T4	Checks for the use of whitespace
C-T5	Checks that the order of modifiers conforms to the Java Language specification
C-T6	Checks for empty blocks
C-T7	Checks that array initialization contains a trailing comma
C-T8	Checks visibility of class members
C-T9	Checks if the code contains duplicate portions of code
C-T10	Restrict the number of &&, || and ^ in an expression

We generated several traces from the execution of the target systems. The idea was to run the systems invoking different features to be able to cover a large portion of the code. This will also allow us to better interpret the results. Table 2, 3 and 4 describe the features that have been traced for each of system.

4.3. Collecting Metrics:

The collection of metrics resulted in a large set of data that we present in Tables 5, 6 and 7. To help interpret the results, we added descriptive statistics such as the average, the maximum, and minimum.

Table 3. Toad Traces

Trace	Description
T-T1	Generates several statistics about the analyzed components
T-T2	Detects and provides reports on uses of Java features, like native code interaction and dynamic reflection invocations, etc.
T-T3	Generates statistics in html format about unreachable classes and methods, etc
T-T4	Specifies bytecode transformations that can be used to generate compressed version of the bytecode files
T-T5	Generates the inheritance hierarchy graph of the analyzed components
T-T6	Generates the call graph using rapid type analysis of the analyzed component
T-T7	Generates an html file that contains dependency relationships between class files

The Call Volume Metrics

Table 5 shows the results of computing the call volume metrics for the Checkstyle system. The initial size metric shows that traces are quite large even when they are triggered using a simple example as input data, which is the case here. As we can see in Table 5, the average size is around 74615 calls. The trace C-T9 is the only trace that does not follow this rule as it generates only 1013 calls. After an analysis of the content of this trace, we found that it is the only one that does not invoke methods of the "antlr" package, which is a package that seems to generate many calls in other traces. Future work should focus on analyzing how the metrics we have defined vary, depending on the system components invoked in traces. The average size of the resulting traces after the repetition removal stage is 33293 calls, which is still too high for someone to completely understand.

Table 4. Weka Traces

Trace	Description
W-T1	Cobweb Clustering algorithm
W-T2	EM Clustering algorithm
W-T3	IBk Classification algorithm
W-T4	OneRClassification algorithm
W-T5	Decision Table Classification algorithm
W-T6	J48 (C4.5) Classification algorithm
W-T7	SMO Classification algorithm
W-T8	Naïve Bayes Classification algorithm
W-T9	ZeroR Classification algorithm
W-T10	Decision Stump Classification algorithm
W-T11	Linear Regression Classification algorithm
W-T12	M5Prime Classification algorithm
W-T13	Apriori Association algorithm

The average size of Toad traces is 220409 calls as shown in Table 6, which is almost three times higher than the average size of Checkstyle traces. The collapse ratio after removing contiguous repetitions, Rcr, is around 5% which is much lower than Rcr for Checkstyle (46%). The average size of the resulting traces is around 10763 calls.

The average size of Weka traces is around 145985 calls. Some Weka algorithms generate much smaller traces such as ZeroR (Trace W-T9). The differences in the size of traces as shown in Table 7 may be due to the complexity of the different data mining algorithms supported by Weka. The average size of the resulting traces after repetition removal is around 16147 calls.

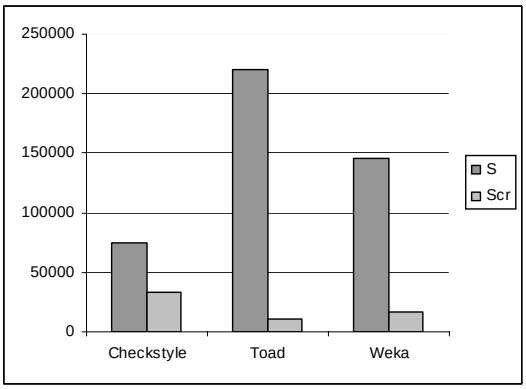

Figure 3. The initial size of traces and their size after the repetition removal stage for the three systems

Figure 3 illustrates the average initial size and the average size after removing contiguous repetitions for traces of the three systems. Although, removal of contiguous repetitions can result in a considerable reduction of the number of calls, the resulting traces of all three systems continue to contain thousands of calls, which is still high for the users of the tools. So repetition removal is necessary but far from sufficient.

The Component Volume Metrics

Table 5 shows that Checkstyle traces involve on average 31% of the system's packages; 16% of the system's classes and 9% of the system's methods.

Tables 6 and 7 show that the other two systems have similar characteristics: The Toad traces involve on average 29% of the system's packages, 19% of the classes, and 12% of the methods; while Weka traces involve 28% of the packages, 10% of the classes and 7% of the methods. Figure illustrates this graphically.

The number of distinct methods (Nm) invoked in traces of all three systems is significantly smaller than the number of calls generated (even after removal of contiguous repetitions). For example, the Checkstyle trace C-T1 invokes 590 methods but generates 37957 calls (after removing contiguous repetitions). This means either that traces contain several sequences of calls that are repeated in a non-contiguous way and/or that several subtrees contain the same methods but are structured in different ways.

The second possibility may pose real challenges for tools that rely on using pattern detection capabilities to help understand traces [2, 8, 9, 13, 14, 15]. The problem is that there might be a need to combine several matching criteria to be able to render useful patterns. This can be hard for the tool users to achieve manually. Tools should be enhanced with the capability to combine several matching criteria in an automatic way. This suggests that the dynamic analysis research community ought to address several issues. First, which matching criteria can be combined and which can not? Secondly, what is the relationship between specific matching criteria and the maintenance tasks at hand? For example, De Pauw [2] suggests a matching criterion that ignores the order of calls when looking for similar sequences of calls. However, this might turn out to be ineffective if the maintenance task at hand consists of fixing a system defect.

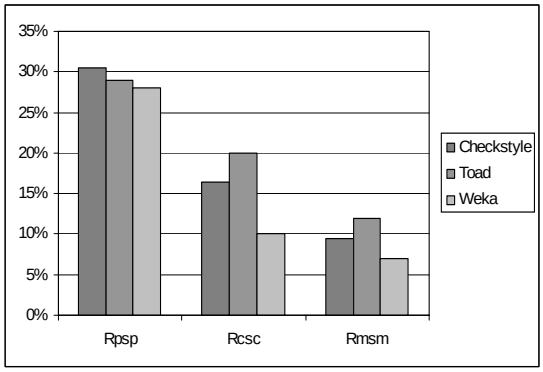

Figure 4. The number of components (packages, classes and methods, respectively) invoked in the traces of the three systems

Comprehension Units and Patterns Related Metrics

Table 5 shows the number of comprehension units and the ratio achieved by transforming Checkstyle traces into acyclic graphs. All Checkstyle traces score a ratio of less than 8% except Trace C-T9 that scores 50%. This is due to the fact that initially this trace is significantly smaller than the other traces in terms of the number of calls but still contains a large number of methods, and that a lower bound on the number of comprehension units is the number of distinct methods in the trace.

Toad traces exhibit a very low ratio as well, which is around 8%. Weka traces exhibit an even lower ratio of 3%. These results confirm the effectiveness of transforming the traces into ordered directed acyclic graphs for trace compression and encoding purposes as

already shown by Reiss et al. [12] and Larus [10]. This can help built scalable tools.

In addition to this, we notice that many of these comprehension units are repeated non-contiguously more than twice in the traces, which qualify them as trace patterns. The average number of patterns that exist in Checkstyle traces is 416, which represents 36% of the number of comprehension units. Toad traces contain in average of 297 (36% of the number of comprehension units), and Weka traces contain on average 77 patterns (33% of the number of comprehension units).

This shows that the number of patterns for large systems can be very high. What is needed is to investigate ways to automatically filter the ones that are important from the ones that are not. This is not supported by most of the existing tools.

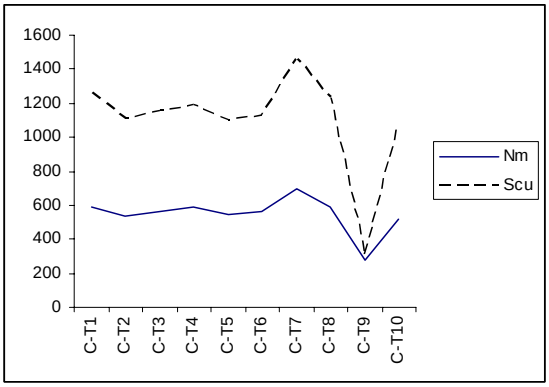

Figure 5. Relationship between the number of methods and the number of comprehension units for Checkstyle – Correlation coefficient = 0.99

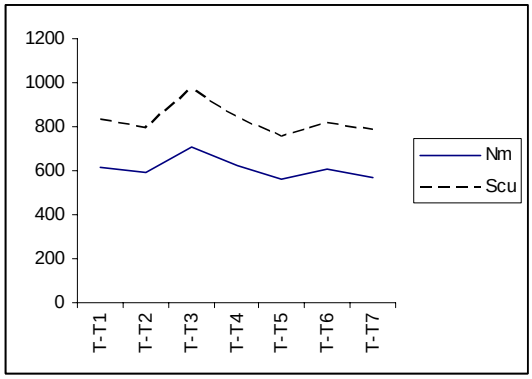

Figure 6. Relationship between the number of methods and the number of comprehension units for Toad – Correlation coefficient = 0.99

Figure 5 illustrates the correlation between the number of methods (Nm) in the trace and the number of comprehension units (Scu) for the Checkstyle system. The graph shows, as expected, that the number of comprehension units (i.e distinct subtrees) of traces depends on the number of methods that are invoked. This also applies to Toad and Weka as illustrated in Figure 6 and Figure 7. Interestingly, the dependency is Weka is somewhat more variable than the other two systems, suggesting that some traces of Weka, and hence the corresponding pieces of functionality are particularly complex (Scu almost equals Nm).

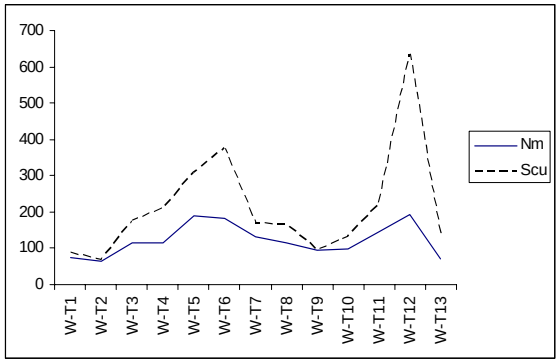

Figure 7. Relationship between the number of methods and the number of comprehension units for Weka – Correlation coefficient = 0.87

5. Applying the Metrics in Tools

As discussed earlier, the goal of the metrics presented in this paper has been to help software engineers understand traces. The metrics will, however, only be useful if actively supported by tool developers.

We suggested that tool developers could, in a rather straightforward way, simply use icons, coloring, or other graphic techniques to show the values of certain of the metrics in order to highlight information about parts of traces.

A key objective would be to display the 'essence' of a trace – just enough to show the software engineer what happened during execution. If, in a visible portion of a trace, Scu (number of comprehension units) is much greater then Nm (number of methods), this suggests that a lot of redundant information is being displayed: Perhaps there are many somewhat similar comprehension units. In this case, we can apply techniques that change the similarity function discussed earlier so that similar comprehension units come to be treated the same, and may then also turn into patterns. Rather than seeing a lot of diversity, the software engineer might then see simply a small set of patterns. The metrics can also be used by tools to help prune the leaves and successively higher branches of traces to make what remains displayed somewhat simpler. We have been investigating ways of removing utilities [6] as one step in this process. However that will often not be enough. A tool could look at the values of ratios such as Rcr and Rtg_{sim}, for each subhierarchy, and based on the values of the ratios, contract the subtree to a certain level.

Table 5. CheckStyle Statistics

Checkstyle	Call Volume Metrics			Component Volume Metrics						Comprehension Units and Patterns Metrics			
Traces	S	Scr	Rcr	Np	Rpsp	Nc	Rcsc	Nm	Rmsm	Scu_{cr}	Rtg_{cr}	Nptt	Rpcu
C-T1	84040	37957	45%	14	33%	114	17%	590	10%	1261	3%	515	41%
C-T2	81052	35969	44%	13	30%	109	16%	540	9%	1106	3%	423	38%
C-T3	81639	36123	44%	13	30%	110	16%	561	10%	1153	3%	439	38%
C-T4	84299	37062	44%	14	33%	117	17%	590	10%	1191	3%	464	39%
C-T5	80393	35455	44%	13	30%	106	16%	547	9%	1098	3%	424	39%
C-T6	81550	36087	44%	14	33%	113	17%	562	10%	1125	3%	437	39%
C-T7	89085	41414	46%	14	33%	148	22%	700	12%	1455	4%	532	37%
C-T8	83106	37163	45%	14	33%	114	17%	586	10%	1234	3%	490	40%
C-T9	1013	618	61%	9	21%	70	10%	276	5%	306	50%	27	9%
C-T10	79969	35083	44%	13	30%	105	16%	521	9%	1071	3%	406	38%
Max	**89085**	**41414**	**61%**	**14**	**33%**	**148**	**22%**	**700**	**12%**	**1455**	**50%**	**532**	**41%**
Min	**1013**	**618**	**44%**	**9**	**21%**	**70**	**10%**	**276**	**5%**	**306**	**3%**	**27**	**9%**
Average	**74615**	**33293**	**46%**	**13**	**31%**	**111**	**16%**	**547**	**9%**	**1100**	**8%**	**416**	**36%**

Table 6. Toad Statistics

Toad	Call Volume Metrics			Component Volume Metrics						Comprehension Units and Patterns Metrics			
Traces	S	Scr	Rcr	Np	Rpsp	Nc	Rcsc	Nm	Rmsm	Scu_{cr}	Rtg_{cr}	Nptt	Rpcu
T-T1	219507	10409	5%	20	29%	172	19%	615	12%	827	8%	293	35%
T-T2	218867	10141	5%	20	29%	169	19%	592	12%	794	8%	282	36%
T-T3	226026	13132	6%	20	29%	191	22%	704	14%	971	7%	347	36%
T-T4	220438	10811	5%	20	29%	177	20%	626	12%	835	8%	299	36%
T-T5	218681	10002	5%	20	29%	164	19%	558	11%	754	8%	271	36%
T-T6	219171	10394	5%	20	29%	170	19%	605	12%	816	8%	296	36%
T-T7	220170	10450	5%	20	29%	165	19%	568	11%	782	7%	288	37%
Max	**226026**	**13132**	**6%**	**20**	**29%**	**191**	**22%**	**704**	**14%**	**971**	**8%**	**347**	**37%**
Min	**218681**	**10002**	**5%**	**20**	**29%**	**164**	**19%**	**558**	**11%**	**754**	**7%**	**271**	**35%**
Average	**220409**	**10763**	**5%**	**20**	**29%**	**173**	**20%**	**610**	**12%**	**826**	**8%**	**297**	**36%**

6. Conclusions and Future Work

In this paper, we developed metrics for the analysis of large execution traces of routine calls. These metrics can be used by tool builders and software engineers who want to better understand traces. Tool builders can use these metrics to tune their techniques to better orient the analyst in his or her quest to understand the trace content. One possible help would be to distinguish parts of the trace that perform complex behavior from parts that are relatively easy to grasp. Researchers can use these metrics to investigate further techniques for reducing the complexity of traces.

Using these metrics, we analyzed traces of three different Java systems to get a better understanding of their complexity. One of our metrics Rcr shows that when we remove contiguous repetitions, the size of the trace is reduced to between 5% and 46% of the original size. However, the resulting traces continue to have thousands of calls, which makes this basic type of collapsing necessary but not sufficient.

We found that traces might cross-cut up to 30% of the system's packages, 22% of the system's classes and 14% of the system's methods. Knowing that traces involve a small number of the system's methods but that these methods generate thousands of calls leads us to the

following observation: Tools which rely primarily on pattern detection will not allow the user to achieve an adequate level of abstraction. The problem is that there might be many subtrees in the trace that contain almost the same methods but structured in different ways. The hard part consists of selecting the appropriate matching criteria that will identify these subtrees as similar. Most tools leave this up to the users, but due to the size and complexity of traces, automated assistance is clearly needed. Tools need to suggest matching criteria that will collapse the trace to a manageable size by pre-computing the effect of each criterion. This computation can be based on the metrics described in this paper.

Another finding regarding patterns is that traces might contain a very large numbers of them: over 500 in the case of the Checkstyle system. These patterns, in turn might have hundreds of occurrences, which can make the understanding of the whole trace using pattern detection a challenging task.

An important area of future work would be to evaluate the usefulness to software engineers of tools that implement these metrics.

There are also many ways to refine the work presented here. Firstly, there is a need to study the different matching criteria for the Scu_{sim} metric. The end goal is to have tools that suggest combining different criteria automatically.

Another idea is to investigate how to distinguish trace components that implement low-level implementation details from the ones that correspond to high-level concepts. The removal of sufficient low-level implementation details should reduce the number of comprehension units to a manageable number. We have started working towards this by studying what characterizes the concept of implementation details that we refer to as *utilities* [6].

We also need to work towards formalizing the concepts presented in this paper. This can be done by defining a formal model to represent traces and metrics leading to the definition of a formal language where various trace metrics can be expressed and related.

Finally, the concept of entropy from information theory can be used to suggest areas of a trace that are more complex. Therefore, a useful avenue of investigation would be to develop trace metrics based on entropy.

Table 7. Weka Statistics

Weka	**Call Volume Metrics**			**Component Volume Metrics**						**Comprehension Units and Patterns Metrics**			
Trace	**S**	**Scr**	**Rcr**	**Np**	**Rpsp**	**Nc**	**Rcsc**	**Nm**	**Rmsm**	**Scu_{cr}**	**Rtg_{cr}**	**Nptt**	**Rpcu**
W-T1	193165	4081	2%	2	20%	10	7%	75	5%	89	2%	28	31%
W-T2	66645	6747	10%	3	30%	10	7%	64	4%	66	1%	15	23%
W-T3	39049	7760	20%	2	20%	12	8%	114	7%	177	2%	39	22%
W-T4	28139	4914	17%	2	20%	10	7%	116	7%	209	4%	49	23%
W-T5	157382	26714	17%	3	30%	19	13%	188	11%	309	1%	120	39%
W-T6	97413	25722	26%	3	30%	23	16%	181	11%	375	1%	137	37%
W-T7	283980	21524	8%	3	30%	15	10%	131	8%	168	1%	76	45%
W-T8	37095	6700	18%	3	30%	13	9%	114	7%	167	2%	41	25%
W-T9	12395	637	5%	2	20%	10	7%	93	6%	96	15%	29	30%
W-T10	43681	6427	15%	2	20%	10	7%	97	6%	131	2%	35	27%
W-T11	403704	34447	9%	4	40%	16	11%	147	9%	220	1%	78	35%
W-T12	378344	54871	15%	5	50%	26	18%	194	12%	637	1%	301	47%
W-T13	156814	9368	6%	2	20%	9	6%	72	4%	134	1%	51	38%
Max	**403704**	**54871**	**26%**	**5**	**50%**	**26**	**18%**	**194**	**12%**	**637**	**15%**	**301**	**47%**
Min	**12395**	**637**	**2%**	**2**	**20%**	**9**	**6%**	**64**	**4%**	**66**	**1%**	**15**	**22%**
Average	**145985**	**16147**	**13%**	**3**	**28%**	**14**	**10%**	**122**	**7%**	**214**	**3%**	**79**	**32%**

References

[1] V. R. Basili, G. Caldiera and H. D. Rombach, "Goal Question Metric Paradigm," *In J. J. Marciniak (ed.), Encyclopedia of Software Engineering 1*, New York: John Wiley & Sons, 1994, pp. 528-532

[2] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, "Execution Patterns in Object-Oriented Visualization", *In Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, NM, 1998, pp. 219-234

[3] J.P. Downey, R. Sethi and R.E. Tarjan, "Variations on the Common Subexpression Problem", *Journal of the ACM*. 27(4), October 1980, pp. 758-771

[4] Checkstyle System. http://checkstyle.sourceforge.net/

[5] A. Hamou-Lhadj, T. C. Lethbridge, "Compression Techniques to Simplify the Analysis of Large Execution Traces", *In Proc. of the International Workshop on Program Comprehension (IWPC)*, Paris, 2002, pp. 159-168

[6] A. Hamou-Lhadj and T. Lethbridge, "Reasoning about the Concept of Utilities", *In Proc. of the ECOOP Workshop on Practical Problems of Programming in the Large*, Oslo, Norway, June 2004

[7] IBM TOAD. http://alphaworks.ibm.com/tech/toad

[8] D. Jerding, and S. Rugaber, "Using Visualization for Architecture Localization and Extraction", *In Proc. of the 4th Working Conference on Reverse Engineering*, Amsterdam, Netherlands, October 1997, pp. 219-234

[9] D. Jerding, J. Stasko, and T. Ball, "Visualizing Interactions in Program Executions", *In Proc. of the 19th ICSE*, Boston, Massachusetts, 1997, pp. 360-370

[10] J. R. Larus, "Whole program paths", *In Proc. of the ACM SIGPLAN '99 conference on Programming language design and implementation*, Atlanta, United States, ACM Press, 1999, pp. 259-269

[11] H. B. Lee, B. G. Zorn, "BIT: A tool for Instrumenting Java Bytecodes", *USENIX Symposium on Internet Technologies and Systems*, Monterey, California, 1997, pp. 73-82

[12] S. P. Reiss, M. Renieris, **"**Encoding program executions", *In Proc. of the 23rd international conference on Software engineering*, Toronto, Canada, pp. 221-230

[13] T. Richner, and S. Ducasse, "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles", *In Proc. of the 18th ICSM*, Montréal, Canada, 2002, pp. 34-43

[14] T. Systä, K. Koskimies, and H. Müller, "Shimba – An Environment for Reverse Engineering Java Software Systems.", *Software Practice & Experience, Volume 31 Issue 4*, 2001, pp. 371-394

[15] T. Systä, "Dynamic Reverse Engineering of Java Software", *In Proc. of the ECOOP Workshop on Experiences in Object-Oriented Reengineering*, Lisbon, 1999, pp. 174-175

[16] Witten I. H., E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 1999